RELAZIONE TECNICA

SVILUPPO DI WEBAPP E MICROSERVIZI IN ARCHITETTURA CLOUD

Giovanni Martucci

M: 1000012435

1. OBIETTIVO

Questo progetto si prepone l'obiettivo di sviluppare una web-app, contente una dashboard di controllo per una visione in tempo reale dei dati estrapolati dai microcontrollori in Arduino. Nello specifico, dall'interfaccia web è possibile richiamare, tramite apposita casella, i dati generati dai microcontrollori per la probabilità di pioggia, per la probabilità di tempo soleggiato e per la temperatura. L'estrapolazione dati avviene tramite i micro-servizi implementati in nodejs, i quali vengono invocati dal back-end collegato all'interfaccia web. Essi intercettano l'ultimo messaggio inviato sul canale dall'Arduino, tramite un'apposita configurazione del protocollo di comunicazione MQTT, in cui vede l'Arduino coinvolto con il ruolo di publisher, dunque pubblica i valori su un canale, mentre i micro-servizi svolgono il ruolo di subscribers al canale. Una volta intercettati i valori richiesti, essi vengono inviati all'interfaccia web che aveva innescato tutte le chiamate in cascata. Di seguito alcuni Framework utilizzati nello svolgimento di tale progetto: Angular, Nodejs, Firebase, AWS, Docker. Per scopo dimostrativo in questo progetto i dati generati dai microcontrollori sono simulati.

2. CENNI SULLA TEORIA

2.1. EC2

Amazon Elastic Compute Cloud (Amazon EC2) è un servizio Web che fornisce capacità di elaborazione sicura e scalabile nel cloud. È concepito per rendere più semplice il cloud computing su scala Web per gli sviluppatori. Mediante l'interfaccia Web service intuitiva di Amazon EC2 è possibile ottenere e configurare la capacità in modo semplice e immediato. L'utente ha il controllo completo delle proprie risorse informatiche che possono essere eseguite nell'ambiente Amazon.

2.2. DOCKER

Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker confeziona il software in unità chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime.

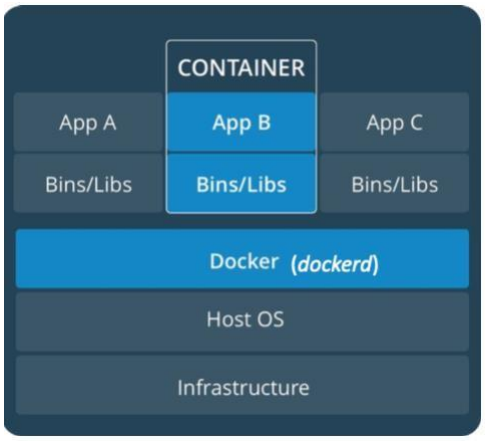

Figura 1 – Docker.

2.3. ANGULAR

Angular è un framework di sviluppo per realizzare applicazioni desktop, web application e mobile application. Permette la realizzazione di applicazioni complesse, sia per desktop che per web o mobile. Utilizzato per lo sviluppo front-end.

Figura 2 – Angular.

2.4.NODEJS – EXPRESS

Node.js è un runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Node.js consente di utilizzare JavaScript per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente.
Il framework Express consente di creare potenti API di routing e di impostare middleware per rispondere alle richieste HTTP, fornendo semplici meccanismi di debugging e una rapida integrazione con vari motori di templating. Qui impiegato come server Nodejs.

Figura 3 – Nodejs.

2.5.FIREBASE

Firebase è una piattaforma serverless per lo sviluppo di applicazioni mobili e web. Esso sfrutta l'infrastruttura di Google e il suo cloud per fornire una suite di strumenti per scrivere, analizzare e mantenere applicazioni cross-platform. Firebase offre infatti funzionalità come analisi statistiche dei dati, database (usando strutture noSQL), messaggistica, autenticazione, segnalazione di arresti anomali, algoritmi di machine learning per la gestione di applicazioni web, iOS e Android.

Figura 4 – Firebase.

3. SPECIFICHE TECNICHE

Di seguito vengono elencate tutte le specifiche tecniche utilizzate per questo progetto:

- Distro: CentOS 7, 8, Ubuntu server 20.4;
- Linguaggio di programmazione: TypeScript, JavaScript, Python;
- Framework utilizzati: Angular12, Nodejs, Express, AWS, Docker, Firebase, Bootstrap;
- Virtualizzazione: VMWare Fusion;

4. MODALITA' D'ESECUZIONE

Ecco un diagramma di flusso dell'esecuzione della web-app (Figura :

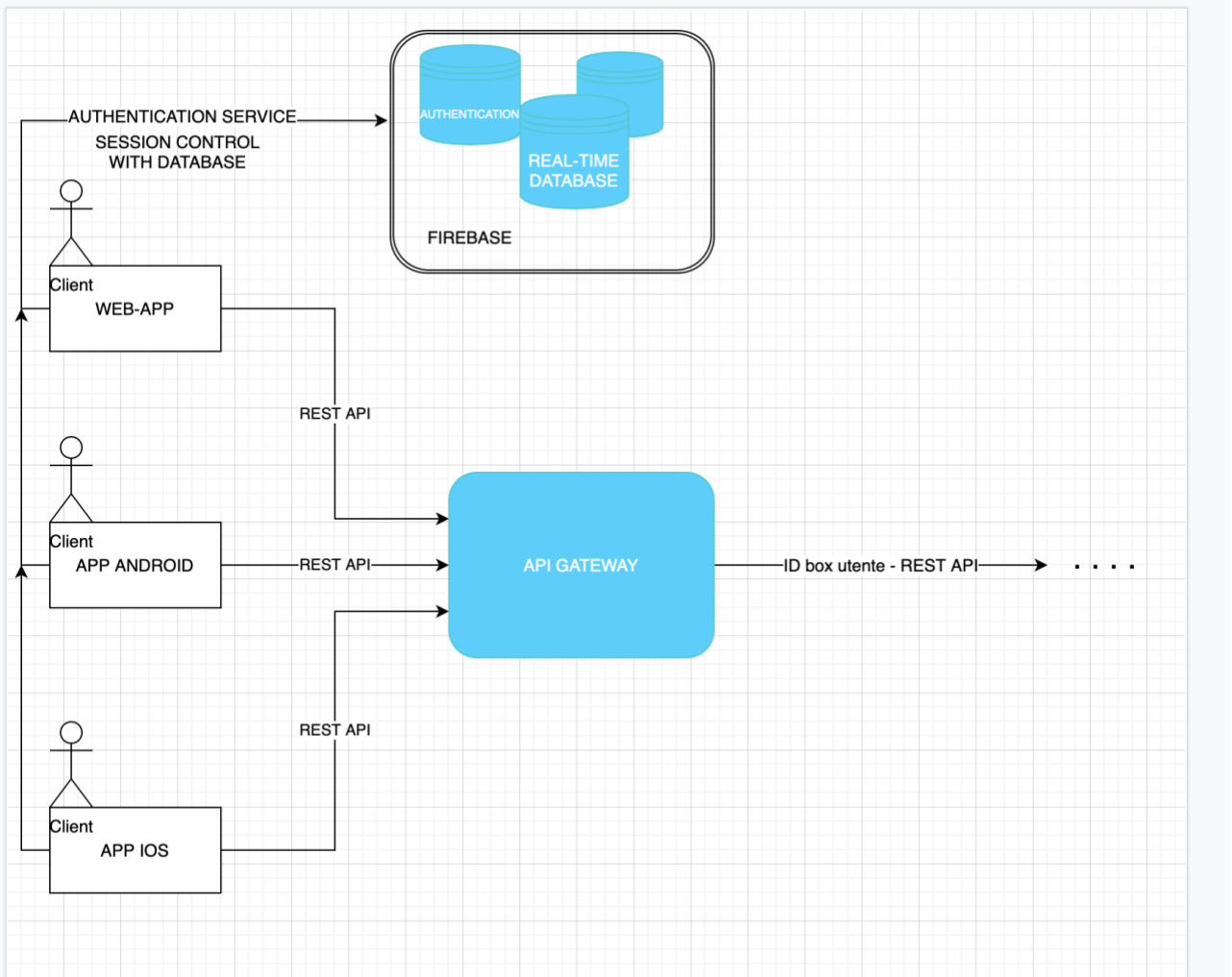

Figura 5 – Diagramma di flusso web-app.

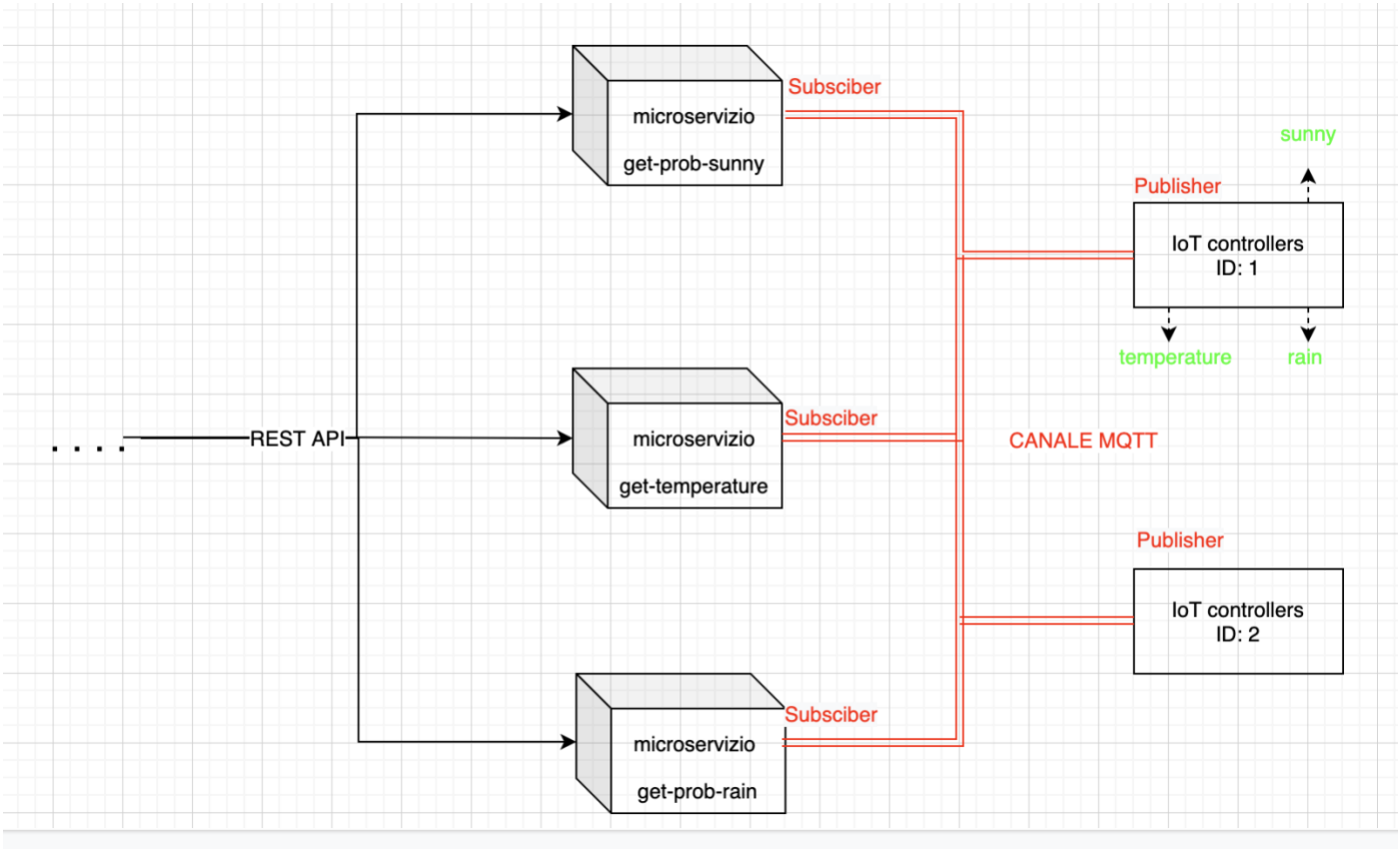

Figura 6 – Diagramma di flusso web-app.

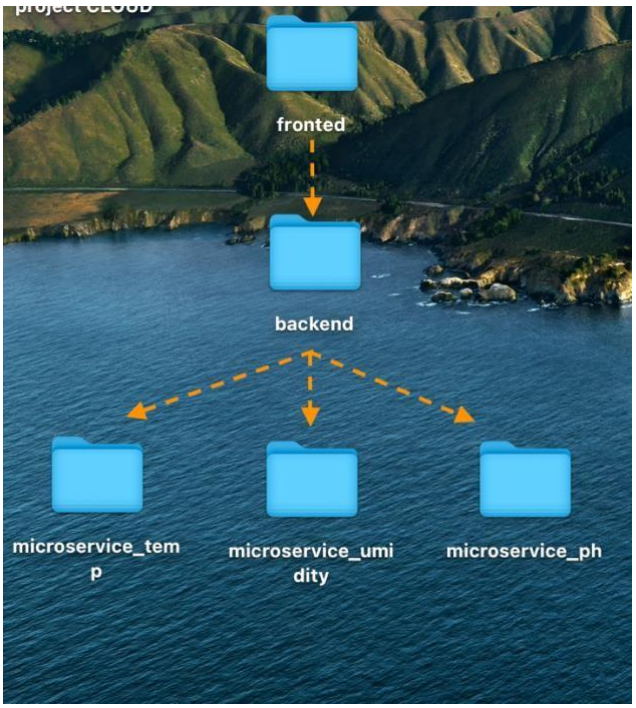

Figura 7 – Schema webapp in cartelle (rappresentanti immagini per container).

Inizialmente ci si è occupati dello sviluppo della web-app strutturata come in figura 5,6,7. Nello specifico è stato utilizzato Angular v.12 per lo sviluppo della parte front-end. In essa sono presenti

varie componenti, tra cui la dashboard di controllo, in cui è possibile visionare i vari valori trasmessi dall'arduino, un componente di login che utilizza il framework Firebase per la gestione dell'autenticazione (tramite l'utilizzo di chiamate API).

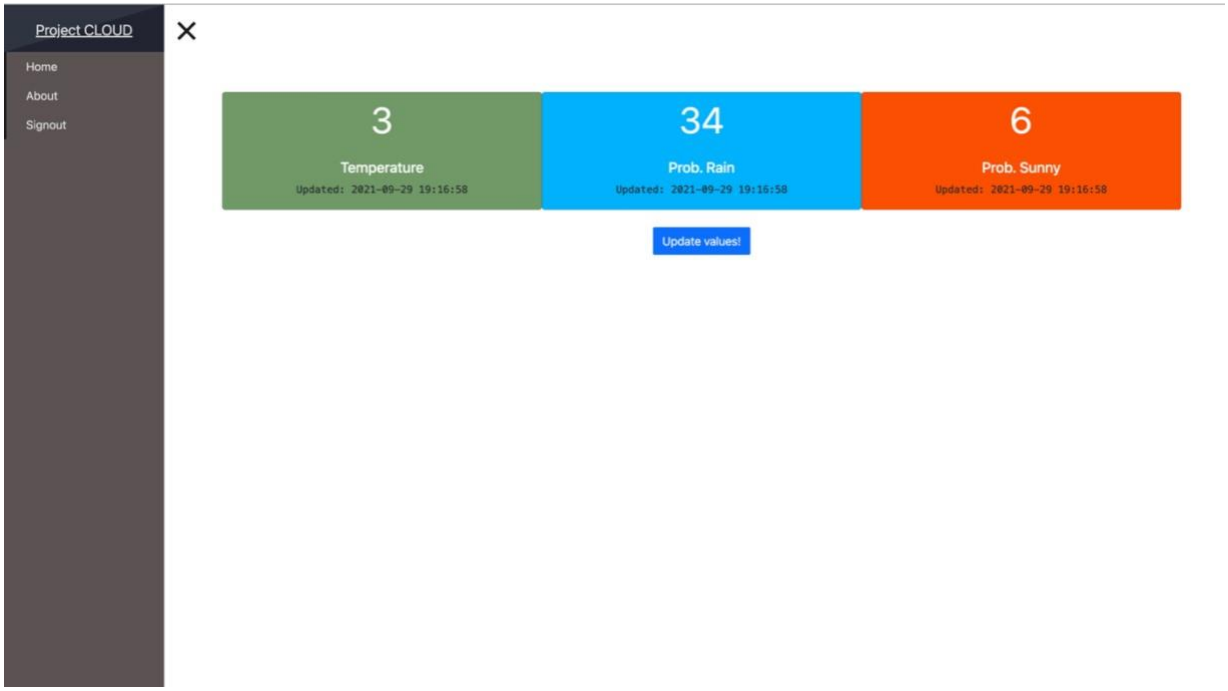

Figura 8 – Componente Dashboard.

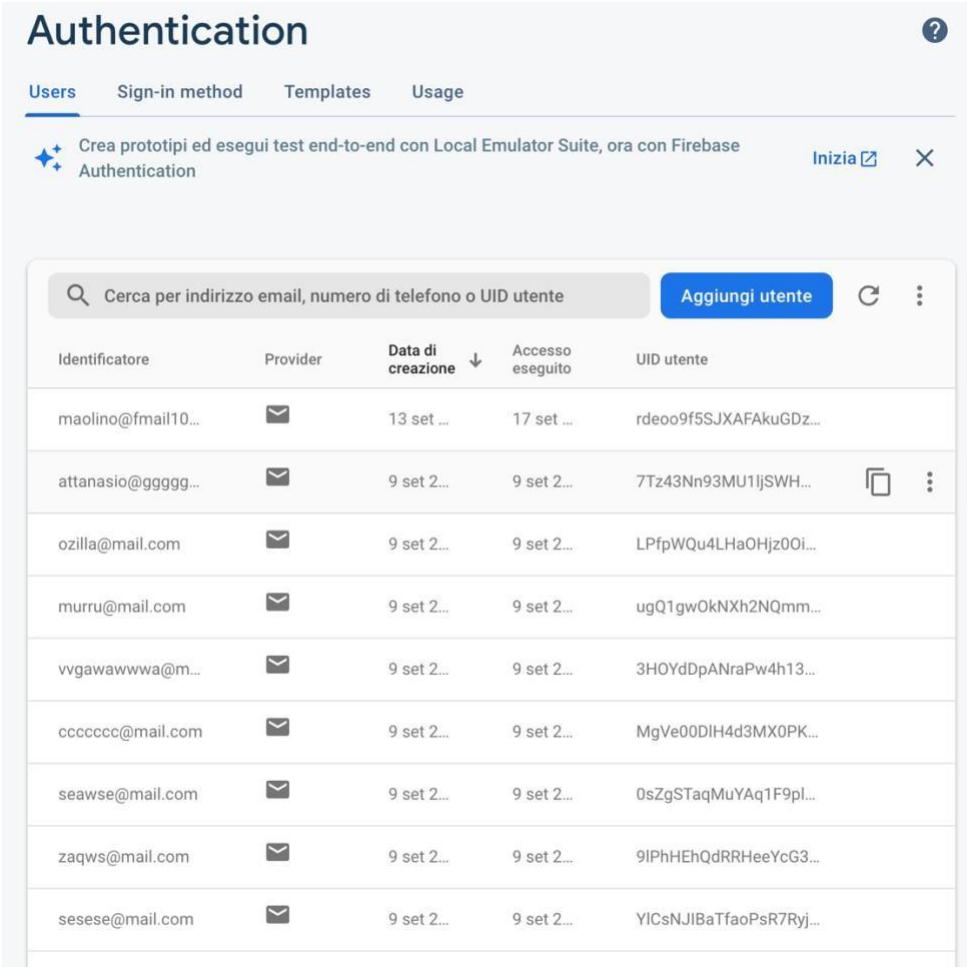

Figura 9 – Firebase servizio Authentication.

Successivamente si è proceduto con lo sviluppo della sessione di login tramite l'utilizzo del Realtime Database, offerto anch'esso dal framework Firebase: al momento di un login l'utente viene inserito nel database e non appena l'utente effettuerà un logout l'utente verrà cancellato dal database, ciò starà ad indicare che la sessione è stata chiusa, dunque se si proverà ad accedere direttamente all'url "/home" si verrà indirizzati alla pagina di login/signup, come definito nelle regole di routing in Angular (app-rounting.module.ts).

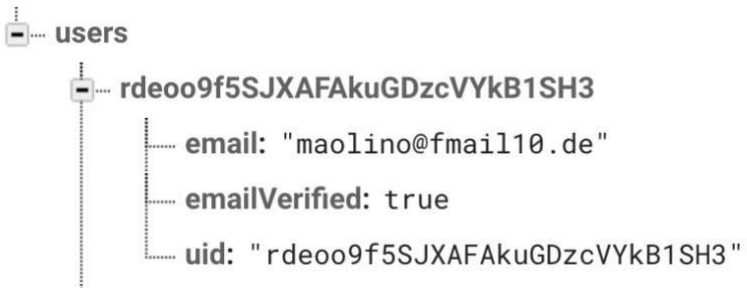

Figura 10 – Firebase Realtime database.

```
import { AuthGuard } from './guard/auth.guard'

const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'home', component: HomeComponent,  canActivate: [AuthGuard] },
  { path: 'register-user', component: SignUpComponent },
  { path: 'forgot-password', component: ForgotPasswordComponent },
  { path: 'verify-email-address', component: VerifyEmailComponent }
];
```

Figura 11 – Regole di routing in Angular.

Come si evince dall'immagine in Figura 11, si può accedere alla home solamente se la variabile AuthGuard) è settata a True (Guard è una metodologia Angular per la corretta gestione delle sessioni degli utenti). Se nel database la sessione dell'utente risulterà attiva, allora questa variabile sarà settata a True e la home diventerà accessibile per quell'utente.

La fase successiva è stata dedicata allo sviluppo della parte back-end in Nodejs ed Express, con la quale si intercettano le richieste dati dal frontend tramite richieste http. All'arrivo di una richiesta dati il back-end effettua a sua volta 3 richieste http ai microservizi che in questo progetto ritornano dei dati simulati, ma nella realtà essi comunicano con l'Arduino che gestisce dei microcontrolli, tramite il protocollo MQTT. In una visione globale del progetto, l'Arduino interpreterà il ruolo di publisher sul canale inviando i valori richiesti dal back-end, mentre i 3 micro-servizi chiamati dal back-end interpreterebbero i ruoli di subscribers, intercettando i valori dal canale in cui sono iscritti. Qui vengono utilizzate apposite funzioni async/await.

4.1. LOCAL IMPLEMENTATION

Una volta completata l'intera web-app, ci si è dedicati allo sviluppo dell'architettura cloud. In un primo momento in locale tramite l'utilizzo di 3 macchine virtuali, istanziate dal software VMWare Fusion, più l'host fisico (MAC OSX).

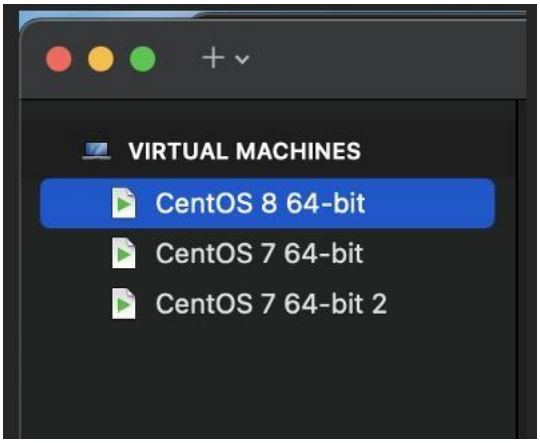

Figura 12 – VMWare Fusion.

Successivamente si è inizializzato il framework Docker. Di seguito tutti i procedimenti utilizzati per la conteinerizzazione della webapp divisa in componenti:

- Sono state create le 5 immagini relative al frontend, backend, microservizio_temperatura, microservizio_probabilità_pioggia, microservizio_probabilità_sole. Ecco i 5 DockerFile stilati per tale operazione:

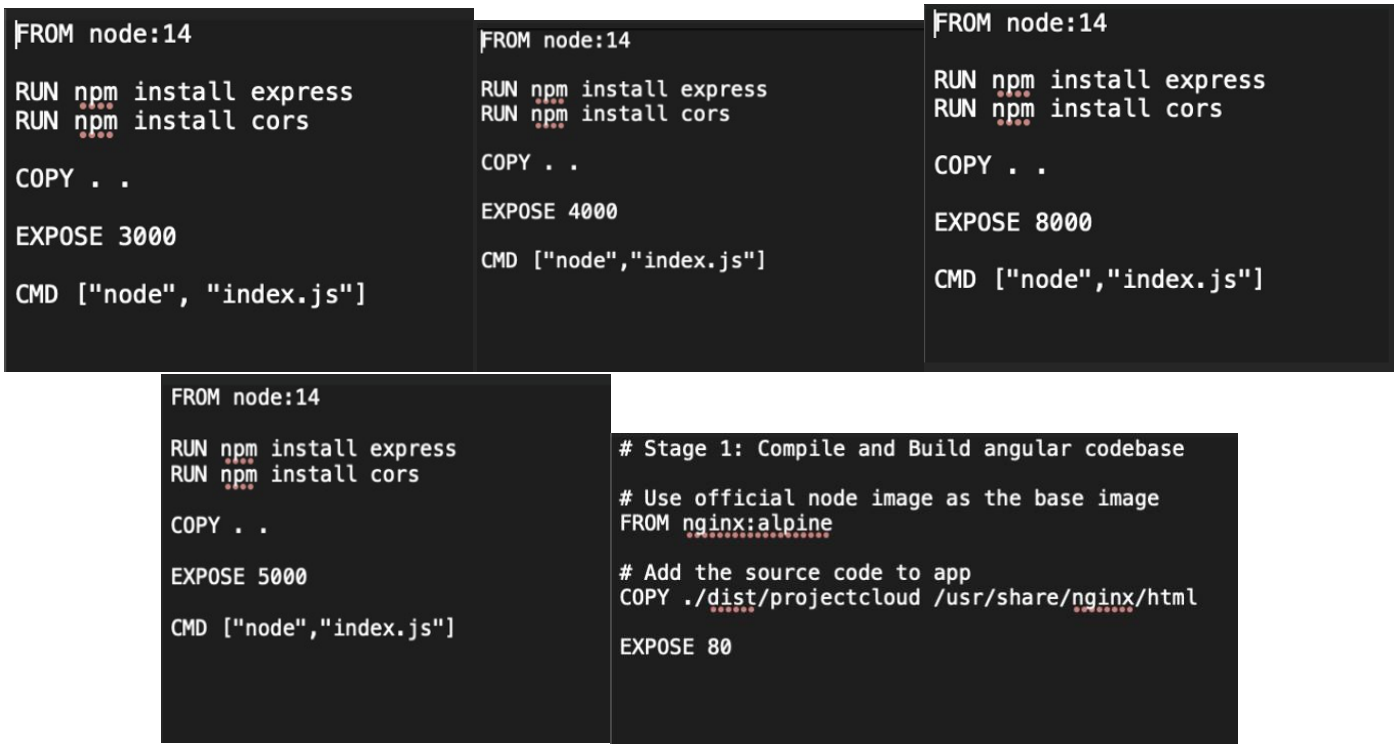

Figura 13 – DockerFile per immagini.

Per la componente Angular (ultima immagine) ci si è avvalsi di un procedimento leggermente diverso per la generazione dell'immagine. Prima di generare l'immagine con docker è infatti necessario buildare manualmente l'applicativo tramite il comando "ng build". Si procede con il comando per generare le immagini: "docker build -t giomar19/frontend:latest ." (Immagini caricate successivamente nella repository relativa all'account docker).

Finora è stato spiegato come è avvenuta la containerizzare delle componenti. Da qui si procede con il deploy dell'app in "produzione" rendendola un servizio, cioè un insieme di container che ascoltano su un unico IP/port rispondendo a turno.

Innanzitutto, tramite il comando "docker swarm init", è stato creato il "docker swarm", ovvero un cluster di host in cui verranno caricati i container. Nello specifico viene nominato come swarm manager l'host chiamante che crea lo swarm. Esso orchestra tutti gli altri node (host) in base a 2 regole:

– emptiest node;

– global node: ogni nodo ottiene un'istanza dell'app (un container)

Tramite il docker mesh lo swarm assicura che ogni nodo dello swarm esponga lo stesso port e assicura un load balacing del tutto trasparente all'utente, permettendo quindi la risposta dello stesso servizio da diversi nodi dello swarm in base ad una strategia che potrebbe essere il round robin.

Successivamente sono stati inseriti gli host creati precedentemente nello swarm come node worker, con il comando:
"docker swarm join --token
SWMTKN-1-04ljxrfy12j5f3yahywrkd7w5aljqklw8hlgg5au5ao1yjsczx-574enhcdgz2j9tf5jz4pu458r
192.168.175.5:2377",
rilasciato alla creazione dello swarm.

Dopo di ciò si è passati al caricamento dei container sullo swarm tramite l'utilizzo del dockercompose.yml, un file in cui vengono dichiarati i servizi che verranno caricati sullo swarm con la scelta del numero di repliche e altre impostazioni, come la subnet, ecc. .

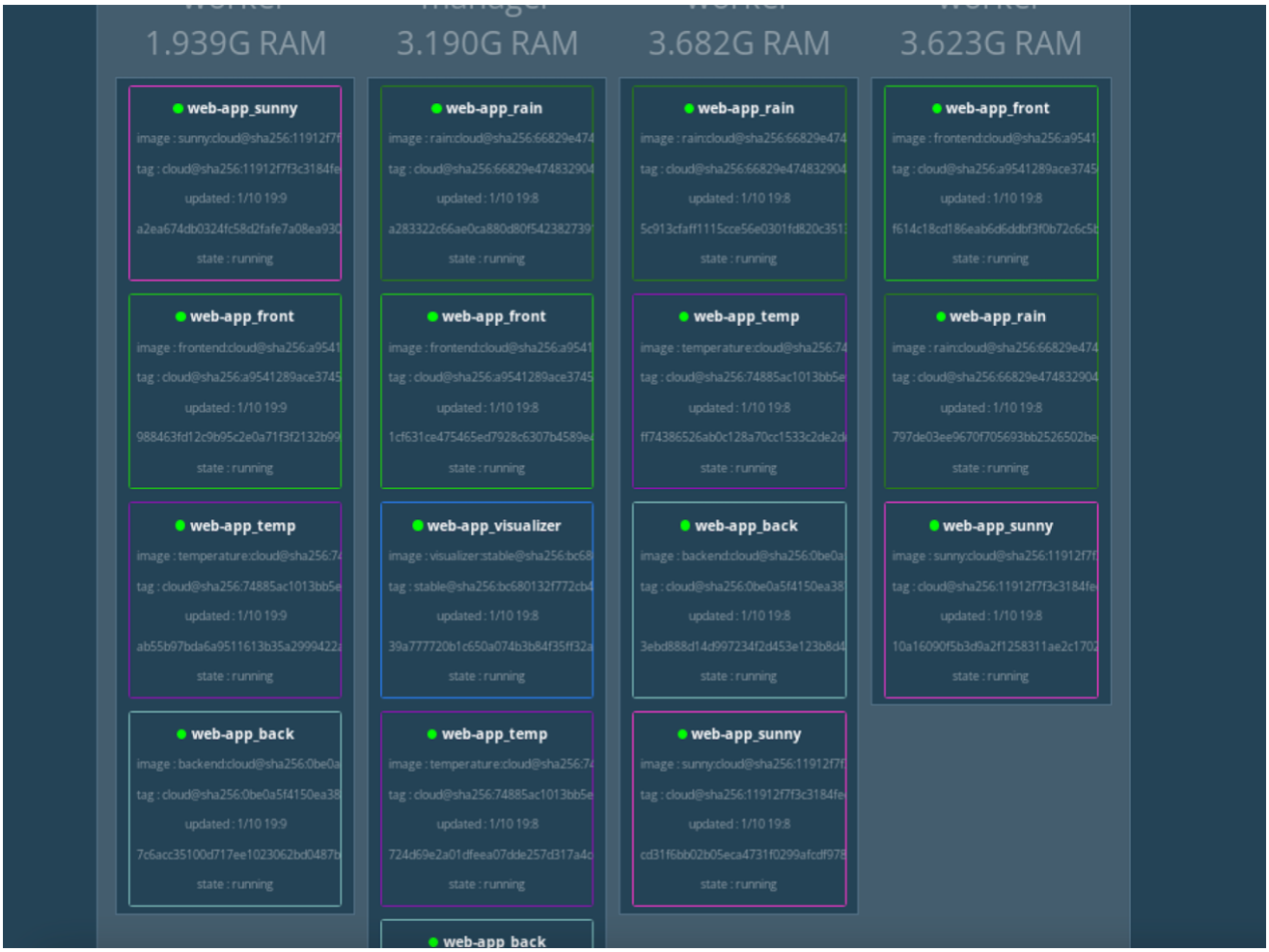

Figura 14 – Immagine "Visualizer" – Swam con 4 host, un manager e tre worker con tre repliche per ciascuna immagine.

4.2.AWS IMPLEMENTATION

Da qui si procede con AWS Cloud:

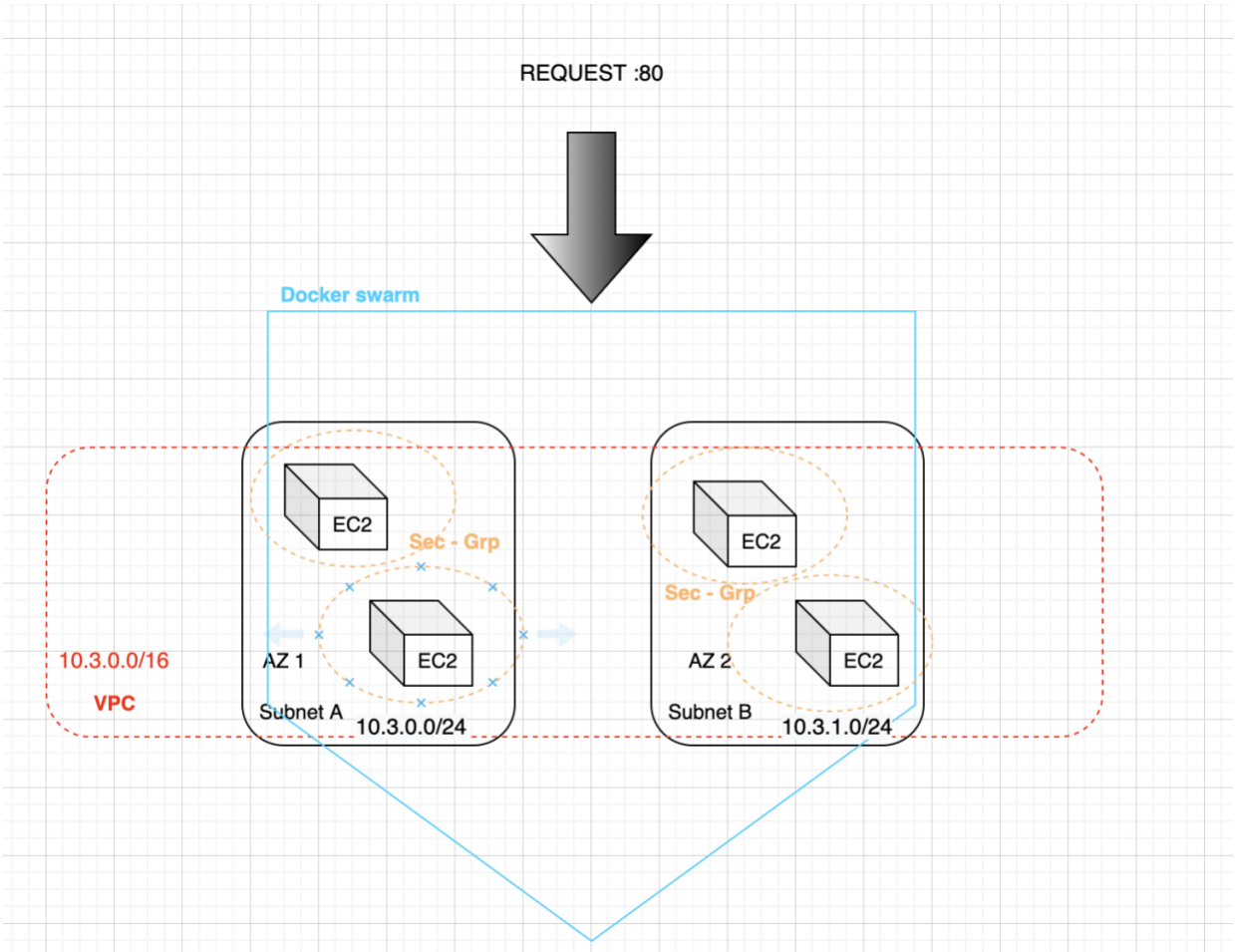

Figura 15 – Architettura Cloud su AWS.

- Sono state istanziate 4 EC2 con distro “ubuntu_server_20.4”.
- Inizializzazione e configurazione dell’intero servizio dedicato al networking: VPC (my-web-app-vpc);

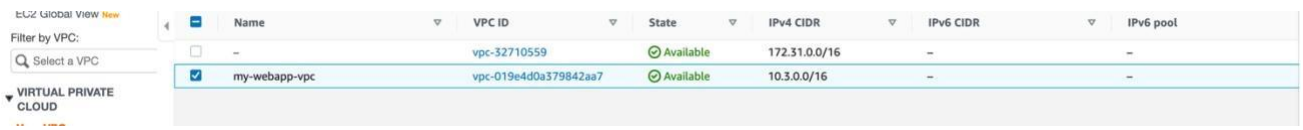

Figura 16 – vpc aws.

- Configurazione dell’Internet Gateway, con aggiunta dell’ip 0.0.0.0/0 sulla route table delle subnet all’interno della vpc (questo permette di poter andare in rete).

Figura 17 – vpc aws.

Figura 18 – inserimento 0.0.0.0 into route table.

- Creazione di 2 Subnet all'interno della vpc (my-web-app-subnet, my-subnet-2);

Figura 19 – Subnet.

- Creazione di una NACL (firewall subnet) che permette il traffic inbound e outbound. Nello specifico sono state abilitate le porte 22, 80, 443 (my-web-app-nacl);

Figura 20 – NACL.

- Security Group (firewall per host) per le machine (my-web-app-sec-group): le porte 7946, 4789, 2377 sono delle porte di default del servizio docker

Inbound rules (12)

Manage tags | Edit inbound rules

Filter security group rules

1

Name	Security group rule...	IP version	Type	Protocol	Port range	Source
–	sgr-058a07a12ddec3c57	IPv6	HTTP	TCP	80	::/0
–	sgr-0059e8667fe06d010	IPv4	Custom TCP	TCP	2377	0.0.0.0/0
–	sgr-0103dc979c562fc65	IPv4	HTTP	TCP	80	0.0.0.0/0
–	sgr-0b78b6c8301723b9f	IPv4	Custom TCP	TCP	3000	0.0.0.0/0
–	sgr-05b184d1c689f6d01	IPv4	HTTPS	TCP	443	0.0.0.0/0
–	sgr-03e389dd166904...	IPv4	Custom TCP	TCP	7946	0.0.0.0/0
–	sgr-0ed5d827dfcfe57f7	IPv4	Custom TCP	TCP	8080	0.0.0.0/0
–	sgr-0b2b6e79abafceea0	IPv6	HTTPS	TCP	443	::/0
–	sgr-00da792b9248a1...	IPv4	SSH	TCP	22	0.0.0.0/0
–	sgr-06812a1691507b...	IPv4	Custom UDP	UDP	7946	0.0.0.0/0
–	sgr-05b04f27b9fe19a35	IPv4	Custom UDP	UDP	4789	0.0.0.0/0
–	sgr-065d380fdaa64f31b	–	All ICMP - IPv4	ICMP	All	sg-030b93737e

Figura 21 – Security Group.

- Inizializzazione di un docker swarm dal nodo manager e inserimento delle macchine nello swarm:

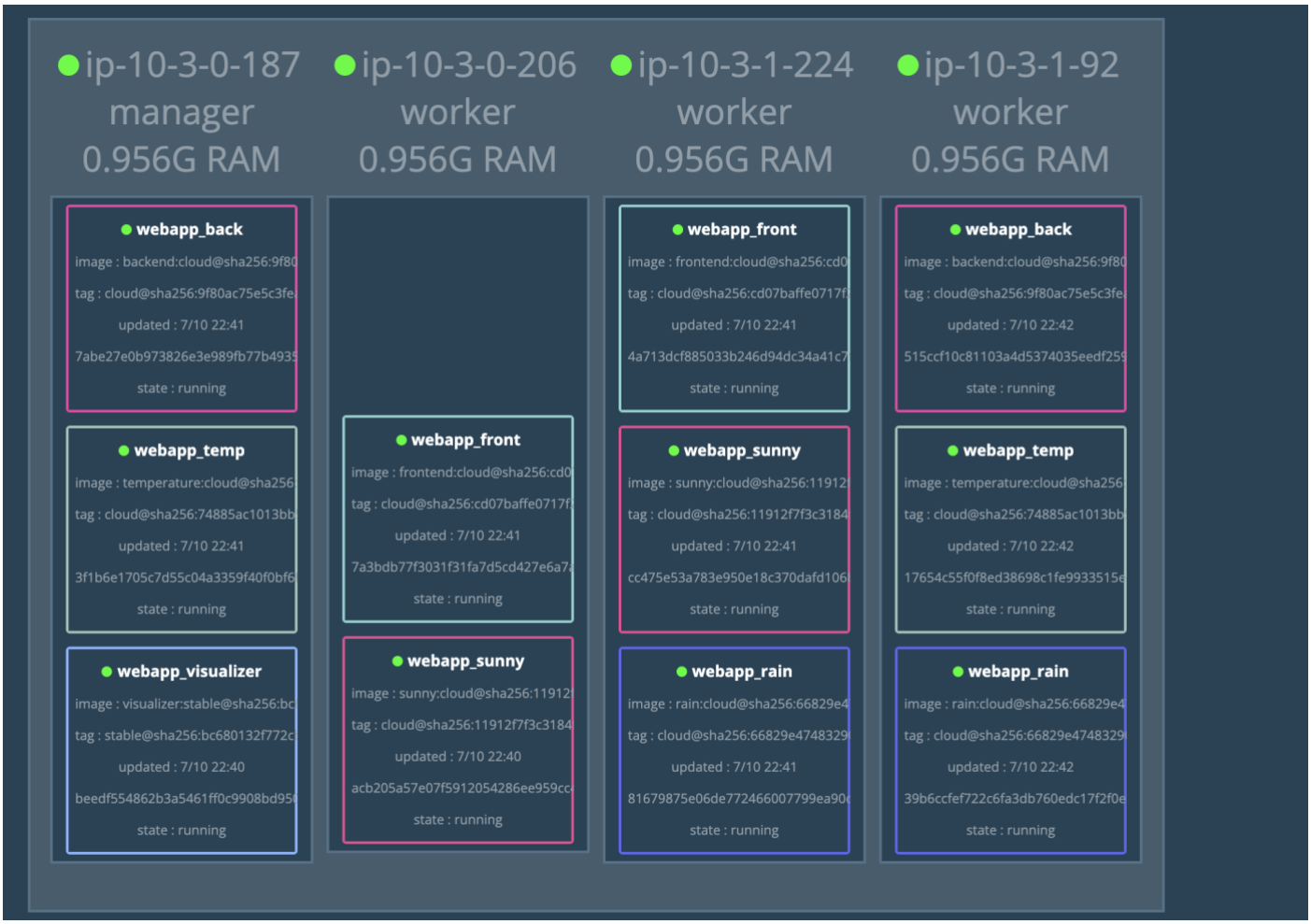

Figura 22 – Docker swarm composto da 4 macchine su due subnet diverse, collogate in due AZ differenti.

Infine il risultato ottenuto dalla configurazione del sistema di networking prevede una segregazione della webapp: sono stati isolati i micro-servizi di estrapolazione dati, così da renderli inaccessibili dal web; gli unici servizi esposti sono il front-end sulla porta 80 e il back-end alla porta 3000, per un possibile utilizzo esterno tramite l'API (in formato json), i restanti servizi invece sono resi inaccessibili tramite apposite configurazioni di NACL E Security Group.

- E' stato creato un profilo IAM denominato "gio.mar", appartenente al gruppo developers, con tali permessi "EC2FullAccess";

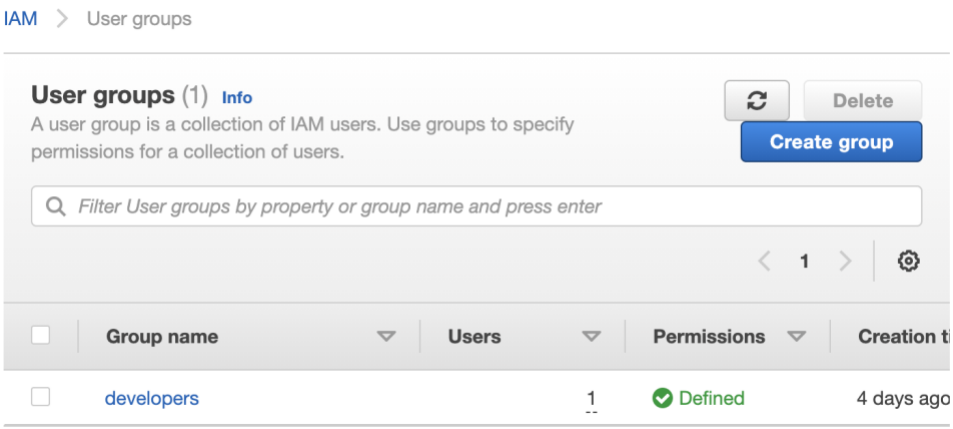

Figura 23 – User Group AWS.

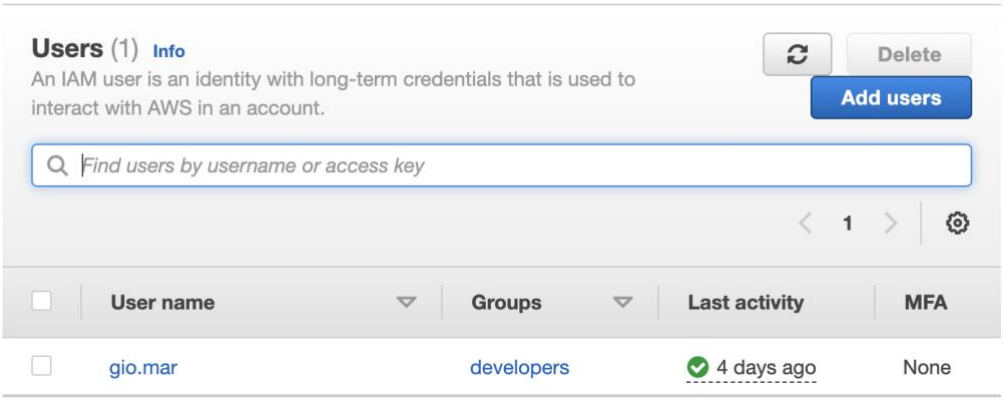

Figura 24 – User AWS che gode della policy generata per il gruppo developers.

- Per ovviare al problema dell'indirizzo ipv4 pubblico in continuo aggiornamento è stato utilizzato un servizio denominato "Dyn DNS" (Figura 20). Questo servizio permette di fissare l'indirizzo ip pubblico con un hostname fisso:

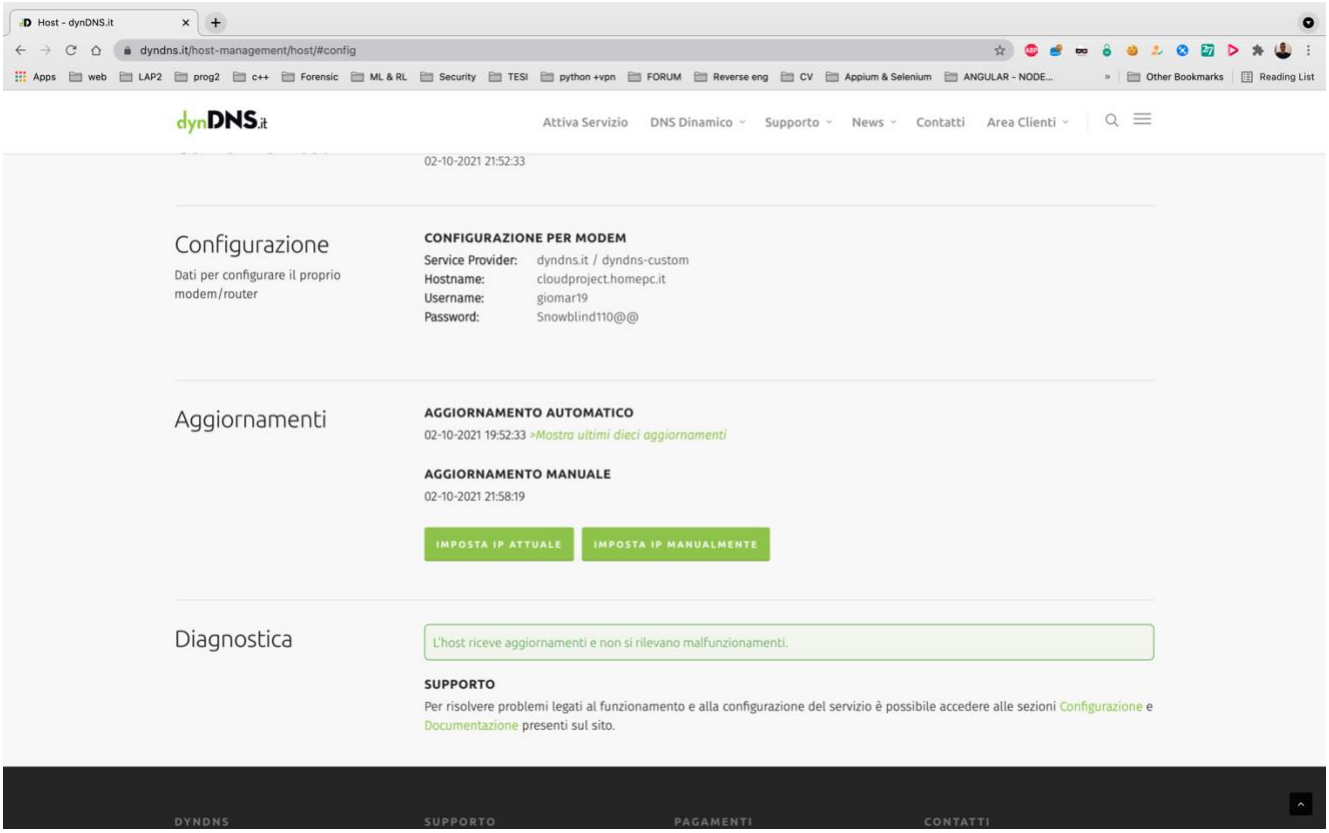

Figura 25 – Dashboard servizio Dyn-DNS.

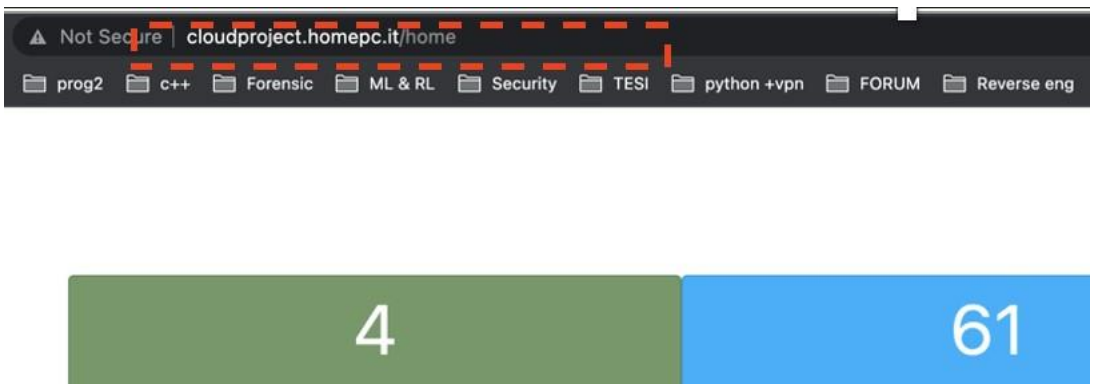

Figura 26 – Accesso effettivo tramite indirizzo IP esterno con l'hostname fornito da Dyn-DNS.

5. RISULTATI

Di seguito la home della webapp che estrapola i dati dai microservizi conteinerizzati e suddivisi nelle istanze EC2 dopo la configurazione dell'intera architettura: qui l'utente può controllare gli effettivi valori dei propri micro-controllori in tempo reale. Oltre all'aggiornamento temporizzato ogni 20 minuti, è possibile cliccare sul bottone di update che invocherà una chiamata al back-end alla porta 3000. Risponderà un'istanza EC2 che conterrà al suo interno uno o più container docker denominati "back-end" (nel sistema vi sono più repliche sparse sui vari nodi dello swarm per ogni tipo di container). Anch'esso invocherà delle chiamate ai microservizi alle porte 4000, 5000, 8000 i quali restituiranno i valori letti (in questo caso simulati) dal microcontrollore in esecuzione.

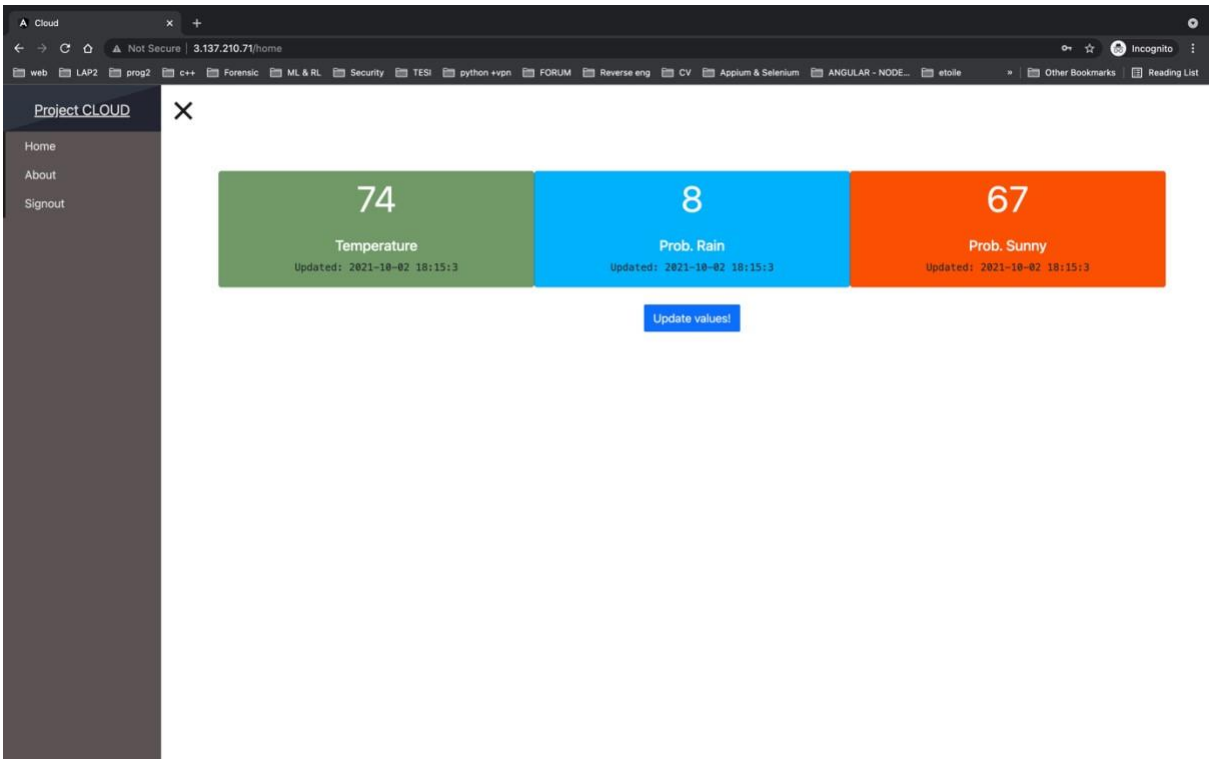

Figura 27 – Accesso tramite indirizzo IP esterno ad EC2 AWS.

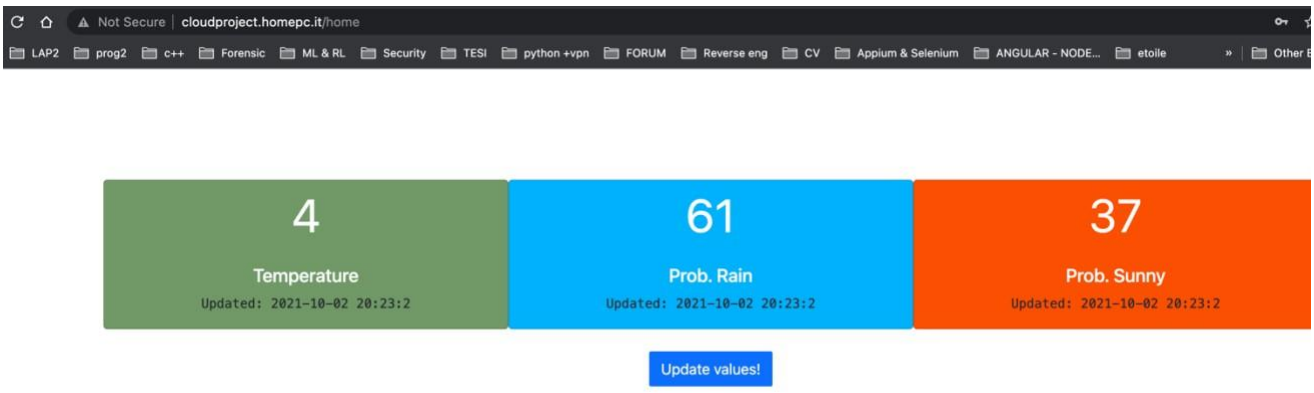

Figura 28 – Accesso tramite indirizzo IP esterno in EC2 AWS con il servizio Dynamic DNS

6. CONCLUSIONI

L’intero progetto ottiene i risultati pianificati: la webapp funziona correttamente all’interno dell’infrastruttura AWS in maniera conteinerizzata. Da tale progetto è stato possibile acquisire e consolidare diverse competenze utilizzate oggi in larga scala nel mondo del lavoro. Si è preso consapevolezza dell’utilizzo e della logica di programmazione distribuita, dell’infrastruttura conteinerizzata, dei vari framework utilizzati per lo sviluppo, come Docker, Kubernetis, AWS, Angular, Nodejs, Firebase, MQTT.

7. SITOGRAFIA

1. https://angular.io/, 20/08/21;
2. https://nodejs.dev/, 21/08/21;
3. https://expressjs.com/it/ 21/08/21;
4. https://firebase.google.com/, 20/08/21;
5. https://aws.amazon.com/it1, 29/09/21;
6. https://aws.amazon.com/it/getting-started/hands-on/deploy-docker-containers/ 29/09/21
7. https://mqtt.org/;
8. https://dyndns.it/;
9. https://kubernetes.io/;

8. REPOSITORY GITHUB

Di segutito il link al progetto Github, in cui è possibile visionare il codice dell'intera webapp (front-end, back-end) e i codici dell'implementazione del sistema dockerizzato:

- https://github.com/Giovanni-Martucci/cloud-project.git